

Sim-alpha: a Validated, Execution-Driven Alpha 21264 Simulator

Rajagopalan Desikan* Doug Burger[†] Stephen W. Keckler[†] Todd Austin[‡]

[†]Department of Computer Sciences *Department of Electrical and Computer Engineering
The University of Texas at Austin

[‡] Department of Electrical Engineering and Computer Sciences
The University of Michigan at Ann Arbor

Department of Computer Sciences
Tech Report TR-01-23
The University of Texas at Austin

ABSTRACT

This technical report describes installation, use, and design of *sim-alpha*, an execution driven Alpha 21264 simulator. To increase simulator accuracy, we have incorporated many of the low level features found in the Alpha 21264. When compared to a hardware 21264 implementation, *sim-alpha* achieves 2% error across a suite of microbenchmarks designed to stress the various microarchitectural features in the simulator. The error across the 10 SPECINT 2000 benchmarks is 6.6% and the 12 SPECFP 2000 benchmarks is 21%, with the net error being 15% across the 22 of the 26 SPEC CPU 2000 benchmarks.

1 Introduction

The computer architecture community relies heavily on simulators to evaluate new ideas. Publicly available simulators like SimpleScalar [1], Rsim [10], Trimaran [5], and SimOS [11] are widely used and shared by researchers, and numerous papers have been published using the results from these tools. However, few of these tools have been compared against actual hardware. In this report, we describe *sim-alpha*, a validated, execution driven, Alpha 21264 processor simulator. *sim-alpha* was written by extending the SimpleScalar [1] tool suite.

sim-alpha models both the implementation constraints, as well as the performance-improving low level features in the 21264. The simulator includes flags which allows the user to enable and disable these features to study their influence. The simulator allows the user to vary the different parameters of the processor such as the issue queue sizes, the fetch width, and the reorder buffer size. *sim-alpha* achieves 2% error across the set microbenchmarks we used for the validation, and 15% across a set of 22 macrobenchmarks from the SPEC CPU 2000 suite. The error across the 10 SPECINT 2000 benchmarks is 6.6%.

The rest of the report is organized as follows. Section 2 describes how to obtain and build *sim-alpha*. Section 3 describes our target system that includes the Alpha 21264 processor, the DS-10L Alphaserver system against which we validate *sim-alpha*, the Digital Continuous Profiling Infrastructure tool set for measuring performance of programs on the native DS-10L system, and the microbenchmarks we used for validating the microarchitecture and memory system in *sim-alpha*. We present *sim-alpha* error across our suite of microbenchmarks and macrobenchmarks in Section 4, and describe usage and internal workings of the tool in Section 5. Finally, Section 6 summarizes our work and suggests future enhancements.

2 Obtaining *sim-alpha*

The *sim-alpha* simulator source code is available as a tar gzipped file through the world wide web at :

```
http://www.cs.utexas.edu/users/cart/code/alphasim-1.0.tgz
```

The microbenchmarks used for the validation can be obtained from :

```
http://www.cs.utexas.edu/users/cart/code/microbench.tgz
```

The SPEC CPU 2000 benchmark binaries can be obtained from :

```
ftp://ftp.simplescalar.org/pub/benchmarks/spec2000/spec2000alpha.tar.gz
```

sim-alpha currently runs only on x86/Linux boxes; Since it does not currently have cross-endian support, it cannot run on big-endian machines. The system call support on *sim-alpha* also currently supports only Linux calls. To build the simulator, uncompress the *tgz* file and type *make* in the resulting *alphasim* directory to build *sim-alpha*

```
tar -xvzf alphasim-1.0.tgz
cd alphasim-1.0
make
```

The `alphasim/tests` directory contains compiled test binaries. The simulator uses the SimpleScalar 3.0 Alpha front end emulator, so it can run any binary compiled for the Alpha ISA. *sim-alpha* takes command line arguments and also accepts arguments in a file. The simulator can be compiled in three modes:

1. *Normal mode* where it includes all Alpha 21264 features. This is the default mode. Type
`make <sim-alpha>`
2. *Flexible mode* where the low level features in the 21264 can be turned on or off. Type
`make flexible`
3. *Functional debug mode* where a functional simulator checks the correctness of the timing simulator. While running in functional debug mode, early instruction retire should be disabled, and only eio traces, introduced with release 3.0 of the SimpleScalar suite, should be used. Type
`make functional`

3 Target Specification

In this section we describe the Alpha 21264 microarchitecture, the Compaq DS-10L Alphaserver machine we used as our reference machine, the Digital Continuous Profiling Infrastructure tool from Compaq which allowed us to measure the performance of programs on the native machine, and the microbenchmarks which helped us isolate errors in *sim-alpha*.

3.1 Alpha 21264 Overview

The Compaq Alpha 21264 [2] [3] [8] [9] microprocessor was introduced in 1998. It implements the Alpha architecture, which is a 64-bit load and store RISC architecture. To operate at high clock frequencies, the 21264 incorporates innovative features such as clustered functional units, merging the branch target buffer with the instruction cache, and using a set-predict cache. In the following subsections, we describe the general features of the microprocessor, as well as some of the low level features, which we have implemented in *sim-alpha*.

3.1.1 Microprocessor features

The 21264 has a seven stage pipeline as shown in Figure 1. The fetch stage of the pipeline fetches a set of four instructions from the instruction cache every cycle. It uses the line predictor to get the address of the instruction to fetch the next cycle. The slot stage of the pipeline statically slots instructions to sub-clusters on which they can execute. The branch predictor also returns with a prediction in this stage. The next stage of the pipeline, the map stage, performs renaming of registers and puts instructions in the issue queues. Instructions issue from the integer and floating-point issue queues in the issue stage, read their input operands in the register read stage, and start executing in the functional unit assigned. The instruction outputs are written back in the writeback stage of the pipeline.

Below we list the main features of the 21264. In *sim-alpha*, all these features can be configured with command line parameters, and the default values are those listed in this section.

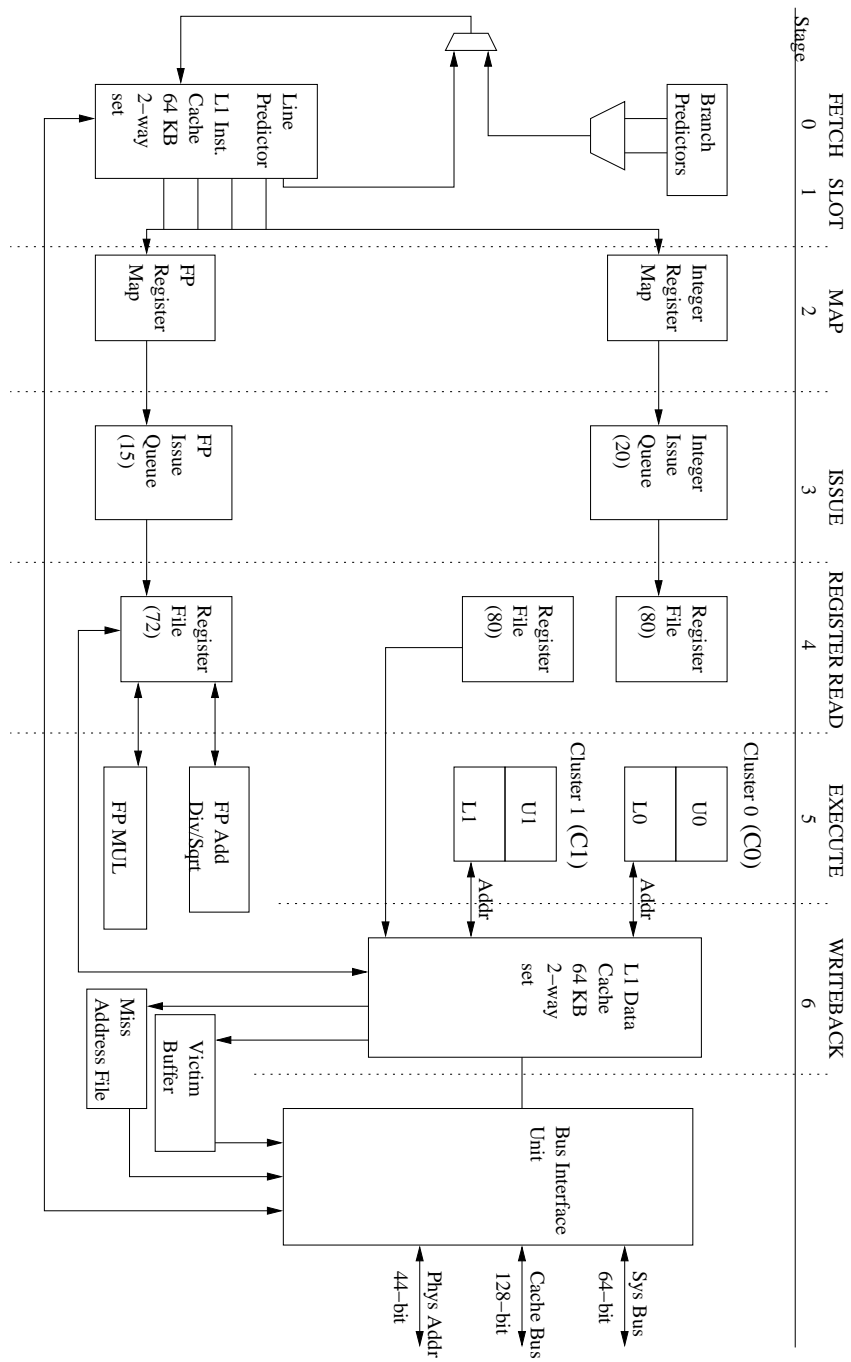


Figure 1: Alpha 21264: Block Diagram (Original diagram courtesy Jim Keller's Alpha 21264 presentation)

- An issue width of six instructions (4 integer and 2 floating point) during each CPU cycle from a 20-entry integer issue queue and a 15-entry floating point issue queue.
- An 80-entry reorder buffer for tracking instructions in flight.
- A demand-paged memory-management unit consisting of a 128-entry, fully-associative data translation buffer (DTB) and a 128-entry, fully-associative instruction translation buffer (ITB).
- Four integer units with an 80 entry register file. These units are called sub-clusters in the alpha, and operate on specific classes of instructions. The 80 entry register file consists of 31 architectural registers, 8 PAL shadow registers, and 41 registers for renaming.
- Two pipelined floating-point units. One unit executes adds, divides, and square roots, and the other unit executes multiplication instructions. The 21264 has 72 floating registers. Of these, 31 are architectural registers, and 41 are used for renaming destination registers of instructions in flight.
- A 64KB virtually addressed instruction cache. The cache is two-way set associative with 64 byte blocks. The 21264 uses a set predictor to choose between the two sets on each access. This ensures single cycle access latency to the I-cache when the set is predicted correctly.
- A virtually indexed, physically tagged dual-read-ported, 64KB data cache. The cache is two-way set associative with 64 byte blocks. The access time for the cache is 3 cycles.
- A tournament branch predictor which consists of
 - a) A two level local predictor that has 1024 entries in the first level (indexed by the PC), with 10 bits in each entry, used to index another 1024 entry table of 3-bit saturating counters.
 - b) A 4096 entry global predictor with 2-bit saturating counters.
 - c) A 4096 entry choice predictor to choose between local and global predictors with 2-bit saturating counters.
- An 8-entry victim data buffer.
- A 32-entry load queue.
- A 32-entry store queue.
- An 8-entry miss address file

3.1.2 Low-level features in the 21264

The following paragraphs describe some of the implementation constraints the designers faced for achieving high clock frequency, and the low-level features they incorporated to achieve high performance.

In the 21264, the branch predictor takes two cycles to make a prediction. This results in a one-cycle bubble between the cycle the instruction is fetched and the cycle the prediction is made. To eliminate this bubble, the 21264 has a line predictor, that effectively acts as a branch target buffer. Each cycle, the line predictor predicts the I-cache line to be accessed in the next cycle. When instructions are fetched from the I-cache, the line prediction bits are also fetched along with

the instructions. These bits are used the next cycle to get the next set of instructions. When the branch predictor completes, the prediction is compared with the line predictor in the slot stage of the pipeline. For certain classes of control instructions like branches and immediate jumps, if the branch predictor prediction differs from the line predictor prediction, fetch is re-initiated with the branch predictor address. The line predictor can store a target for a set of four instructions. In *sim-alpha*, using command line parameters, we can vary the number of instructions for which the line predictor stores a prediction. We can also disable the line predictor, and use a regular btb instead.

The instruction cache is two-way set-associative. To achieve single cycle access, a way predictor in the I-cache predicts which set is being accessed in the current cycle. Way prediction gives the effective access time of a direct mapped cache, although it does result in a 2-cycle bubble on a set misprediction. The way predictor latency can be varied in *sim-alpha*.

In the map stage, the processor does not know the number of free registers available to rename in the current cycle. Hence, it ensures that there are always enough registers available to rename for the next two cycles by stalling for 3 cycles, whenever the number of free physical registers falls below 8. After 3 cycles it again evaluates the number of free physical registers, and will stall again for 3 cycles if the free register condition is still unsatisfied.

The integer execution core is partitioned into two clusters C0 and C1. Each cluster has a copy of the 80-entry physical register file, and two sub-clusters called lower (L) and upper (U), containing the integer functional units. These sub-clusters are not symmetric, and contain different numbers and types of functional units. For example, an integer multiply functional unit is present only in U1. The register files contain identical values. There is a one-cycle delay to transfer data from one cluster to another. Thus, dependent instructions can issue during successive cycles only to the same cluster, and will have to wait one cycle to issue to the other cluster. The 21264 statically slots instructions to the two sub-clusters in the slot stage to achieve a better load balance, and then dynamically chooses the cluster during issue. For example, if a fetched octaword contains an `add`, a `mult`, a `load`, and a `shift` instruction in that order, the slot stage will slot it as LULU to ensure maximum usage of execution resources. *sim-alpha* provides command line options for varying the number of clusters, disabling slotting and clustering, and for setting the value of the cross cluster delay.

The D-cache in the 21264 has a 3-cycle hit latency. To facilitate faster instruction wakeup on a cache hit, the 21264 uses a technique called load-use speculation, where it issues instructions dependent on the load assuming a load hit. If the load misses in the cache, these instructions are squashed and reissued. In *sim-alpha*, we approximate load-use speculation by reissuing only the instructions that are dependent on the missing load.

The 21264 also uses prefetching on an I-cache miss to improve performance. The 21264 can prefetch four instruction cache lines from the L2 cache on an I-cache miss. Four lines is also the default prefetch value in *sim-alpha*. However, the number of lines to prefetch can be varied using command line parameters. The 21264 has an 8-entry unified victim buffer to cache recently evicted blocks from the I-cache, D-cache, and L2 cache. *sim-alpha* caches blocks only from the level-one D-cache in the victim buffer. The size of the victim buffer can be varied in *sim-alpha*.

The branch predictor uses an adder to precompute targets of immediate branches. This adder enables the 21264 to predict targets of immediate branches correctly even if the line predictor is wrong. The 21264 also has a mechanism called early instruction retire to detect no-ops early in the pipeline (map stage). These no-ops are retired immediately, and thus do not consume execution resources. The user can enable or disable the adder and the early instruction retire in *sim-alpha*.

To enforce correct memory accesses, the 21264 uses order traps. Order traps result in the pipeline being flushed, and the instruction being restarted from the fetch stage of the pipeline. There are two main types of order traps: Load-Load order traps and Store-Load order traps. The 21264 invokes a load trap on a newer load instruction that has been issued before an older load instruction to the same address. To detect a store trap, the 21264 compares the addresses of all store instructions as they are issued to loads in the load queue. If the processor detects a newer load to the same address in the load queue, it invokes a store trap on the newer load. Store traps are necessary to ensure that loads and stores to the same address happen in program order. Traps are expensive in terms of performance, the minimum cost being 12 cycles. Hence the 21264 uses special hardware to reduce the occurrence of store traps. The processor has a 1024 one bit table called the *stWait* table, indexed by the PC, to stall issue of loads causing order traps. This bit is fetched from the I-cache with each instruction. The processor does not issue a load for which the *stWait* bit is set, until all previous stores have issued. On a store trap, this bit is set for the faulting load when it is re-fetched. All bits in the *stWait* table are unconditionally cleared every 16,384 cycles. The 21264 also has another type of trap called Mbox trap, for ensuring correctness in the memory system. The Mbox traps also result in a flushing of the pipeline but are triggered by events occurring in the memory system such as outstanding misses to two loads to same address but different destination registers, outstanding misses to different physical addresses that map to the same D-cache or L2 cache line, and store queue overflow. The Load-Load order traps and the Mbox traps can be disabled in *sim-alpha*. The user can also set the size of the *stWait* table.

3.2 Compaq DS-10L Alphaserver

We used the Compaq DS-10L Alphaserver to validate *sim-alpha*. The workstation has a single 21264 processor clocked at 466 MHz, a 2 MB external L2 cache (direct mapped, with 64 byte blocks), and 256 MB of physical memory. The workstation runs version 5.1 of Compaq Tru64 UNIX. The DS-10L has custom memory controller chips which consists of a single control chip, the Digital DC1046C, and two chips which act as data switches, the Digital DC1047B. The SDRAM consists of 16 chips of 8MB each running at 125 MHz and an 8-ns access time. The C compiler on the DS-10L is version 6.3-025 of the Compaq C compiler.

3.3 Digital Continuous Profiling Infrastructure

The Digital Continuous Profiling Infrastructure (DCPI) [4] for Compaq Alpha platforms permits continuous low-overhead profiling of entire systems, including the kernel, user programs, drivers, and shared libraries. DCPI (subsequently renamed Continuous Profiling Infrastructure) samples the Alpha performance monitoring counters to collect information about each program running on the system.

DCPI can be used for measuring the frequency of certain events on the Alpha 21064, 21164, and the 21264. On the 21264, DCPI can measure the number of cycles taken by a program, number

of instructions retired, Mbox traps incurred, number of retired itlb misses, number of single and double dtlb misses, number of retired conditional branches, and number of retired unaligned traps

DCPI calculated the number of cycles taken by the programs to complete on the native DS-10L system. This number can then be compared against the number of cycles taken in the simulator, to compute the simulator error.

3.4 Microbenchmarks

Figure 2 gives a brief description of the microbenchmarks we used for validating *sim-alpha*. The first row lists the microbenchmarks we used for testing the front-end such as the line predictor implementation and the branch predictor implementation. The *C* in front of the names of these microbenchmarks signifies that these test control flow. The second row lists the microbenchmarks for testing the execution core such as the scheduler. The *E* in the microbenchmark name stands for the execution core. The last row lists the microbenchmarks for testing the memory system parameters such as the level-one cache latency, the level-two cache latency, and the main memory latency. The *M* here stands for the memory system. A more complete description of the microbenchmarks can be found in [6] [7]. The source code for all the microbenchmarks can be obtained from the website listed in Section 2.